On the Prediction of Smart Contracts' Behaviours

Cosimo Laneve, Claudio Sacerdoti Coen, Adele Veschetti

On the prediction of smart contracts' behaviours

Cosimo Laneve[0000−0002−0052−4061]⋆, Claudio Sacerdoti Coen[0000−0002−4360−6016], and Adele Veschetti[0000−0002−0403−1889]

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
{cosimo.laneve, claudio.sacerdoticoen, adele.veschetti2}@unibo.it

Abstract. Smart contracts are pieces of software stored on the blockchain that control the transfer of assets between parties under certain conditions. In this paper we analyze the bahaviour of smart contracts and the interaction with external actors in order to maximize objective functions. We define a core language of programs with a minimal set of smart contract primitives and we describe the whole system as a parallel composition of smart contracts and users. We therefore express the system behaviour as a first logic formula in Presburger arithmetics and study the maximum profit for each actor by solving arithmetic constraints.

1 Introduction

Smart contracts are programs that run on distributed networks with nodes storing a common state in the form of a blockchain. These programs are gaining more and more interest because they implement the so-called *decentralized applications*, which are applications that can handle and transfer assets of considerable value (usually, in the form of cryptocurrency like Bitcoin). Several decentralized applications have already been applied to asset management scenarios ranging from food supply chain management to energy market management and to identity notarization. The smart contracts of such applications are written in programming languages that are targeted to different blockchains. Two such languages are Solidity for Ethereum (which is imperative) [7] and Liquidity for Tezos (which is functional) [14].

Decentralized applications consist of smart contracts and users, such as humans performing either computer actions or physical ones. Since they run on systems that have no coercing central authority, the uncertainty of the overall emerging behaviour is very high and this is a critical issue when asset movements are at the core of applications. Therefore it becomes important to understand the protocols between interacting parties and, when possible, use smart contracts to regulate behaviours of users that systematically try to maximize their revenues or to minimize losses. For example, a client behaves in different ways in order to minimize the cost κ of a good (e.g. he may choose one company or another). On the other hand, the interacting company tries to maximize its revenue; therefore

⋆ Research partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)".

it strives for the greatest value κ such that the client has still a convenience in acquiring its own good. Determining the least value κ is complex because it may not only depend on the price, but also on the trademark, the delivery type, etc.

In this paper, to suitably address the foregoing issues, (*i*) we adhere to a formal modelling approach, (*ii*) define an analysis technique and (*iii*) prototype the verification process. A precise account of the work follows.

As regards the formal modelling, since (human) users and smart contracts act concurrently and independently, we adopt methods and techniques from the domain of process algebras. As such, we depart from most of the literature on application of formal methods to smart contracts that study their properties as sequential programs. In Section 2 we introduce a unified calculus of actors – both contracts and users, the `scl` calculus – that is expressive enough (it is Turing complete) and features method invocations, field updates, conditional behaviour, recursion and failures. According to the semantics of `scl`, systems, which are parallel compositions of smart contracts and users, perform *transactions*, e.g. sequences of smart contract operations that are triggered by users. Transactions may return a value or may fail; in the first case the states of smart contracts that have taken part in the transaction are committed; in the second case the states backtrack to the last committed one. In parallel to transactions, users may evolve internally in a nondeterministic way (on the contrary, smart contracts' behaviours are deterministic). The model of `scl` is a transition system that enables symbolic analysis of properties – see Section 3. In particular, transitions retain two informations: one is a standard label, say μ, highlighting the action performed, the other one, say ψ, is a formula that records the choices and the guards of conditionals. The two labels play different roles in our analysis technique.

Given the model of a `scl` program, in Section 4 we define an objective function as a map from labels μ to integer expressions. For example, such function may return 1 if the label has a given type, or it may return some expression on the symbolic names occurring in the label. In general we are interested in determining computations that maximize or minimize the sum of the values of an objective function on their labels and in selecting strategies that allow users to behave correspondingly. Once this is done, we analyze whether tuning up and down the symbolic names may generate more profit or reduce loss for one interacting party. To this aim, we select a sensible state $\mathcal{S}$ and the corresponding transition system rooted at $\mathcal{S}$ (we assume there is no cycle and that the transition system is finite). By means of the labels μ and ψ of the transitions, we define a first order logic formula, called the *characteristic formula*, that summarizes the transition system describing concisely the values of the objective function for every possible run.

When the model is Presburger (a decidable fragment of arithmetics where formulas contains only integer numbers, equality, strict inequality, addition and multiplication by a constant), the characteristic formula belongs to an extension of Presburger arithmetics that can be decided via quantifier elimination. The formula without quantifiers allows us to reason about strategies that bring to

goals with higher values (e.g. maximize the profit) for each actor by solving arithmetic constraints. The general cases of infinite, acyclic models are addressed in our technique by analyzing finite unfoldings.

We are currently terminating the implementation in OCaml of a tool that, given a set of contracts, an initial state and an objective function, automatically extracts the open model, computes the characteristic formula and applies quantifier elimination over it. This elimination step also checks whether the formula is satisfiable, i.e. it detects the reachable final states and computes the set of values of the objective function that can be observed in runs that lead to them. These sets are represented as linear mappings over domains that are union of polytopes, i.e. solutions of a systems of linear inequations in normal form. The inequations constrain the choices that users can take according to those of other users or to external inputs to the system. Maximizing linear functions over linear inequations is mathematically trivial.

We conclude in Section 5 by discussing future research directions.

Related works. In the past few years formal methods have been largely used to analyze smart contracts with the aim of verifying the security of potentially dangerous compositions with untrusted codes. One of the most cited motivation has been the famous `TheDAO` attack [15] that stole several million dollars during a crowdfunding procedure and caused an hard fork in the Ethereum blockchain.

An initial contribution is [2], which proposes an analysis framework based on a compilation of Solidity to `F*`, a functional language aimed at program verification with a powerful type and effect system. Using `F*` types, they are able to trace Ethers (the Ethereum cryptocurrency) and discover critical patterns in smart contracts. A different technique has been followed by [10] and [11], sticking to symbolic execution. Similarly to our technique, they use symbolic values for inputs and study symbolic computations by mean of the formula that accumulates the constraints on the inputs. This formula is different from our characteristic formula in Section 4. In particular, while our formula describes *every possible computation* and we use constraints on symbols to determine values that maximize some quantity, in [10] and [11], they are interested in discovering critical patterns of a *single computation*.

In the same line of verifying and validating smart contracts, the contribution [3] combines formal methods and game theory to analyze protocols that also involve players (human users) with different/competing gaming strategies. The technique is the following: game theory is used to analyze the behaviour of the players in the protocol, then the resulting strategies are modelled in a probabilistic system for automated validation. Our approach is somehow the opposite: we define the overall system (or part of it) in a formal model and derive the strategies by analyzing the model. In this paper, we stick to a discrete model (the choice operator in users' behaviours is not probabilistic), leaving to future research the extension to stochastic models.

Another contribution that is close to our one is [4]. In this case, the authors define a simplified language for smart contracts that is loop-free. Then they pro-

vide an automatic translation into stateful concurrent games and analyze these games by means of interval abstraction that is demonstrated to be sound. The technique is very powerful and of practical relevance, considering that models of concurrent games are very large. Unlike to this work, our modelling technique is based on process algebra and our analysis relies on Presburger arithmetics formulae. The evaluation of the practical relevance of our technique is postponed to future research.

2 The calculus of smart contracts

In this section we define a core language of programs featuring a minimal set of smart contract primitives, such as method invocations, field updates, conditional behaviour, recursion and failures.

We use a countable set of *variables*, ranged over by x, y, z, a countable set of *smart contract names*, ranged over by a, b, c, and a countable set of *user names*, ranged over by id, id', id''. Smart contract names and user names are generically addressed by α, α', ... and we assume they are partitioned into disjoint sets such that names of a same set belong to *a same class* and those in different sets belong to *different classes*. The property that a name α belongs to a class $\mathtt{C}$ is expressed by $\alpha \in \mathtt{C}$.

Classes $\mathtt{C}$ have the form $\mathtt{C} : (\overline{F}, \overline{M})$ where $\overline{F}$ is a sequence of *field definitions* $\mathtt{T\ f}$, $\overline{M}$ is a sequence of *method definitions* $\mathtt{T\ m}(\overline{\mathtt{T}\ x})\{\ s_m\ \}$ with $\overline{\mathtt{T}\ x}$ and s_m respectively being the *formal parameters* and the *body* of $\mathtt{m}$. In the whole paper, we assume that sequences of declarations $\overline{T\ x}$ and method declarations $\overline{M}$ do not contain duplicate names. *Types* $\mathtt{T}$ are either naturals *Nat* or names α. Hereafter we write $\overline{k}$ for possibly empty, finite sequences $k_1, \ldots, k_n$ of various entities.

The syntax of statements, rhs-expressions and expressions is given in Figure 1. A statement s may be either a return of an expression, or a field update (plus a continuation), or a conditional or the nondeterministic choice $s + s'$. We assume that bodies of smart contract methods (i) do not have nondeterministic choice (they are deterministic) and (ii) do not have expressions with fail, except for return fail.

A rhs-expression z may be either an expression e or a *synchronous* method invocation. An expression e may be either a standard expression or fail. In e op e', op is a standard operation on naturals. We assume that method bodies of users also have the expression $\mathtt{isfailed}(e)$ that returns 1 if the value of e is fail, 0 otherwise.

The semantics of `scl` statements is defined by a transition relation

$$\alpha : s, \ell \xmapsto[\psi]{\mu} s', \ell'$$

where s and s', with an abuse of notation, are *runtime statements*, ℓ and ℓ' are *memories*, e.g. maps from field names to values v, μ is either empty or $v.\mathtt{m}(\overline{v'})$, and ψ is a formula. The transition means that executing s in an actor α with a memory ℓ amounts to produce an action μ and a formula ψ and executing s' in

Syntax:

$$
\begin{array}{rcll}
s & ::= & \mathsf{return}\ e \mid \mathsf{this}.\mathtt{f} = z; s \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid s + s & \text{(statements)} \\
z & ::= & e \mid e.\mathtt{m}(\overline{e}) & \text{(rhs-expressions)} \\
e & ::= & \mathit{naturals} \mid x \mid \alpha \mid \mathsf{fail} \mid \mathsf{this} \mid \mathsf{this}.\mathtt{f} \mid e\ \mathsf{op}\ e & \text{(expressions)}
\end{array}
$$

States and runtime statements:

$$
\begin{array}{rcll}
\ell & ::= & [\cdots, \mathtt{f} \mapsto v, \cdots] & \text{(memories)} \\
v & ::= & \mathit{naturals} \mid \alpha \mid \mathsf{fail} & \text{(values)} \\
s & ::= & \cdots \mid \mathtt{0} \mid id \mid id[v] \mid \alpha.\mathtt{f} = \bullet; s \mid s; s \mid s +^{x} s & \text{(runtime statements)}
\end{array}
$$

runtime statement transition $\overset{\mu}{\underset{\psi}{\longmapsto}}$ (μ may be empty or α; ψ is a formula):

$$
[\textsc{upd}]\ \frac{[\![e]\!]_{\alpha,\ell} = v}{\alpha : \alpha.\mathtt{f} = e; s, \ell \underset{\mathtt{true}}{\longmapsto} s, \ell[\mathtt{f} \mapsto v]}
\qquad
[\textsc{meth}]\ \frac{[\![e]\!]_{\alpha,\ell} = v \quad [\![\overline{e'}]\!]_{\alpha,\ell} = \overline{v'} \quad \mathsf{fail} \notin v, \overline{v'} \quad v \in \mathtt{C} \quad \mathtt{m}(\overline{x}) = s_{\mathtt{m}} \in \mathtt{C} \quad s'' = s_{\mathtt{m}}\{^{\overline{v'}, v}/_{\overline{x}, \mathsf{this}}\}}{\alpha : \alpha.\mathtt{f} = e.\mathtt{m}(\overline{e'}); s, \ell \overset{v.\mathtt{m}(\overline{v'})}{\underset{\mathtt{true}}{\longmapsto}} s''; \alpha.\mathtt{f} = \bullet; s, \ell}
$$

$$
[\textsc{meth-fail}]\ \frac{[\![e]\!]_{id,\ell} = v \quad [\![\overline{e'}]\!]_{id,\ell} = \overline{v'} \quad \mathsf{fail} \in v, \overline{v'}}{id : id.\mathtt{f} = e.\mathtt{m}(\overline{e'}); s, \ell \underset{\mathtt{true}}{\longmapsto} s, \ell[\mathtt{f} \mapsto \mathsf{fail}]}
\qquad
[\textsc{return}]\ \frac{[\![e]\!]_{\alpha,\ell} = v}{\alpha : \mathsf{return}\ e; s, \ell \underset{\mathtt{true}}{\longmapsto} s[v], \ell}
$$

$$
[\textsc{if-true}]\ \frac{[\![e]\!]_{\alpha,\ell} \neq 0}{\alpha : \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s'; s'', \ell \underset{\mathtt{true}}{\longmapsto} s; s'', \ell}
\qquad
[\textsc{if-false}]\ \frac{[\![e]\!]_{\alpha,\ell} = 0}{\alpha : \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s'; s'', \ell \underset{\mathtt{true}}{\longmapsto} s'; s'', \ell}
$$

$$
[\textsc{fix}]\ \frac{x\ \text{fresh}}{id : (s_1 + s_2); s, \ell \underset{\mathtt{true}}{\longmapsto} (s_1 +^{x} s_2); s, \ell}
\qquad
[\textsc{choice}]\ \frac{i \in \{1, 2\}}{id : (s_1 +^{x} s_2); s, \ell \underset{x=i}{\longmapsto} s_i; s, \ell}
$$

Fig. 1. Syntax and runtime statement semantics of `scl`.

ℓ'. Actions μ are commitments to the context, formulas ψ are essential for our analysis in the next sections.

Runtime statements, as reported in Figure 1, extend statements with `0` representing termination, id recording the user that initiated the transaction, $id[v]$ returning a value to the user id, $\alpha.\mathtt{f} = \bullet; s$ representing a continuation waiting for a value that will replace the symbol $\bullet$, with $s; s'$ denoting the sequential composition, and $s +^{x} s'$, an alternative form of $s + s'$ that retains the fresh variable to be used for recording the choice (this information is necessary in the analysis of Section 4). Sequential composition is considered associative.

The following auxiliary functions are used in the semantic rules:

– $\ell[\mathtt{f} \mapsto v]$ is the *memory update*, namely $(\ell[\mathtt{f} \mapsto v])(\mathtt{f}) = v$ and $(\ell[\mathtt{f} \mapsto v])(\mathtt{g}) = \ell(\mathtt{g})$, when $\mathtt{g} \neq \mathtt{f}$.

– $s[v]$ is the *delivery of a value v to a runtime statement s*, namely

$$s[v] \stackrel{\text{def}}{=} \begin{cases} \alpha.\mathtt{f} = v\,;s' & \text{if } s = \alpha.\mathtt{f} = \bullet\,;s' \quad \text{and } v \neq \mathsf{fail} \\ id.\mathtt{f} = \mathsf{fail}\,;s' & \text{if } s = id.\mathtt{f} = \bullet\,;s' \quad \text{and } v = \mathsf{fail} \\ id[\mathsf{fail}] & \text{if } s = a.\mathtt{f} = \bullet\,;s'\,;id \quad \text{and } v = \mathsf{fail} \\ id[v] & \text{if } s = id \end{cases}$$

It is worth to observe that $(\alpha.\mathtt{f} = \bullet\,;s')[\mathsf{fail}]$ behaves in different ways according to α being a user or a smart contract. In the first case the field of the user is updated with fail – which is possible for users –, in the second case the whole statement fails and the failure is reported to the user that triggered the whole transaction.

– $[\![e]\!]_{\alpha,\ell}$ is a partial function that returns the value of e. The value of fields of α is retrieved in the memory ℓ of α: $[\![\alpha.\mathtt{f}]\!]_{\alpha,\ell} = \ell(\mathtt{f})$. The function is undefined if e tries to use fields of an actor $\neq \alpha$. We omit the definition of $[\![e]\!]_{\alpha,\ell}$ when e is an operation, but we require that it must be fail when one of the arguments is fail. $[\![\overline{e}]\!]_{\alpha,\ell}$ returns the tuple of values of $\overline{e}$.

Let us comment some semantic rules in Figure 1. Rule [UPD] defines the semantics of a field update: the expression e is evaluated in an actor α with a memory ℓ; the resulting memory binds the value to the field $\mathtt{f}$. Rule [METH] defines method invocations $\alpha.\mathtt{f} = e.\mathtt{m}(\overline{e'})\,;s$ when the evaluation of e and $\overline{e'}$ does not return a failure. In this case, the method dispatch is performed by using the value v of the carrier (because every name belongs to a class) and the statement to evaluate becomes the instance of the body of $\mathtt{m}$, followed by the update $\alpha.\mathtt{f} = \bullet$, where $\bullet$ represents a place-holder, and the continuation s. The transition is labelled $\stackrel{v.\mathtt{m}(\overline{v'})}{\longmapsto}_{\mathtt{true}}$ meaning that we are invoking the method $\mathtt{m}$ of actor v with actual parameters $\overline{v'}$. Rule [METH-FAIL] addresses failures in the evaluation of expressions of a method invocation; in this case the invocation is not performed and the field is updated with fail. Rule [RETURN] defines the semantics of return e. There are two types of return continuations s: one is $\alpha.\mathtt{f} = \bullet\,;s'$ – see rule [METH] –, the other one is id (this will be clear in the semantics of `scl` programs). Additionally, the semantics depends on whether α is a smart contract or a user, and on whether the value v is a failure or not. To manage all these cases we use the auxiliary function $s[v]$. The semantics of conditionals is standard. On the contrary, the semantics of nondeterminism is not standard and deserves few comments. First of all, nondeterminism may only occur in user codes, therefore the actor here is id. Then, we need to keep track of the nondeterministic choices in order to study the behaviour of smart contract programs. To this aim we use a fresh variable x each time a choice is about to be performed and $(s_1 + s_2)\,;s$ transits into an intermediate statement $(s_1 +^x s_2)\,;s$. In turn this statement becomes either $s_1\,;s$ or $s_2\,;s$ and the choice is recorded by letting $x = 1$ or $x = 2$ in the formula labelling the transition, respectively.

States:

$$\mathsf{S} ::= \quad s \quad | \quad a(\ell \cdot \ell') \quad | \quad id(\ell, s) \quad | \quad \mathsf{S} \mid \mathsf{S} \qquad \text{(states)}$$

Auxiliary functions $commit(\cdot)$ and $backtk(\cdot)$ (the two functions are homomorphic with respect to $\mid$):

$$\begin{array}{lll} commit(s) = s & commit(a(\ell \cdot \ell')) = a(\ell \cdot \ell) & commit(id(\ell, s)) = id(\ell, s) \\ backtk(s) = s & backtk(a(\ell \cdot \ell')) = a(\ell' \cdot \ell') & backtk(id(\ell, s)) = id(\ell, s) \end{array}$$

State transition $\xrightarrow[\psi]{\mu}$ (μ may be empty, α, $\checkmark$ or fail; ψ is a formula):

$$\frac{a : s, \ell \overset{}{\underset{\psi}{\longmapsto}} s', \ell'}{a(\ell \cdot \ell'') \mid s \underset{\psi}{\longrightarrow} a(\ell' \cdot \ell'') \mid s'} \text{[SC-MOVE]} \qquad \frac{id : s, \ell \overset{}{\underset{\psi}{\longmapsto}} s', \ell'}{id(\ell, s) \underset{\psi}{\longrightarrow} id(\ell', s')} \text{[ID-MOVE]}$$

$$\frac{a : s, \ell \overset{a'.\mathtt{m}(\overline{v})}{\underset{\psi}{\longmapsto}} s', \ell \quad a \neq a'}{a(\ell \cdot \ell') \mid a'(\ell'' \cdot \ell''') \mid s \underset{\psi}{\longrightarrow} a(\ell \cdot \ell') \mid a'(\ell'' \cdot \ell''') \mid s'} \text{[INVK]} \qquad \frac{a : s, \ell \overset{a.\mathtt{m}(\overline{v})}{\underset{\psi}{\longmapsto}} s', \ell}{a(\ell \cdot \ell') \mid s \underset{\psi}{\longrightarrow} a(\ell \cdot \ell') \mid s'} \text{[INVK-SELF]}$$

$$\frac{id : s, \ell \overset{a.\mathtt{m}(\overline{v})}{\underset{\psi}{\longmapsto}} s''; id.\mathtt{f} = \bullet; s', \ell \quad \bullet \notin s''}{id(\ell, s) \mid a(\ell' \cdot \ell') \mid \mathsf{0} \underset{\psi}{\longrightarrow} id(\ell, id.\mathtt{f} = \bullet; s') \mid a(\ell' \cdot \ell') \mid s''; id} \text{[INVK-SC]} \qquad \frac{id : s, \ell \overset{id.\mathtt{m}(\overline{v})}{\underset{\psi}{\longmapsto}} s', \ell}{id(\ell, s) \underset{\psi}{\longrightarrow} id(\ell, s')} \text{[INVK-ID-SELF]}$$

$$\frac{v \neq \mathsf{fail}}{id(\ell, s) \mid id[v] \overset{\checkmark}{\underset{\mathtt{true}}{\longrightarrow}} id(\ell, s[v]) \mid \mathsf{0}} \text{[END-OK]} \qquad id(\ell, s) \mid id[\mathsf{fail}] \overset{\mathsf{fail}}{\underset{\mathtt{true}}{\longrightarrow}} id(\ell, s[\mathsf{fail}]) \mid \mathsf{0} \text{ [END-FAIL]}$$

$$\frac{\mathsf{S} \overset{\checkmark}{\underset{\psi}{\longrightarrow}} \mathsf{S}'}{\mathsf{S}'' \mid \mathsf{S} \overset{\checkmark}{\underset{\psi}{\longrightarrow}} commit(\mathsf{S}'') \mid \mathsf{S}'} \text{[CMT]} \qquad \frac{\mathsf{S} \overset{\mathsf{fail}}{\underset{\psi}{\longrightarrow}} \mathsf{S}'}{\mathsf{S}'' \mid \mathsf{S} \overset{\mathsf{fail}}{\underset{\psi}{\longrightarrow}} backtk(\mathsf{S}'') \mid \mathsf{S}'} \text{[BKT]} \qquad \frac{\mathsf{S} \underset{\psi}{\longrightarrow} \mathsf{S}'}{\mathsf{S}'' \mid \mathsf{S} \underset{\psi}{\longrightarrow} \mathsf{S}'' \mid \mathsf{S}'} \text{[TAU]}$$

Fig. 2. Semantics of `scl`.

2.1 Semantics of `scl` programs

A *smart contract program* is a pair $(\mathcal{D}, \mathsf{S})$, where $\mathcal{D}$ is a *finite set* of *class definitions* and S is *state*. States, as defined in Figure 2, are *parallel composition* of *actors* $a(\ell \cdot \ell')$, called *smart contracts*, or *actors* $id(\ell, s)$, called *users*, and *exactly one* runtime statement, called *blockchain-statement*. Smart contracts have *pairs of memories* $\ell \cdot \ell'$ where ℓ is the *current memory* and ℓ' is the *last committed memory*. In case of commits, the current memory ℓ becomes the last committed memory; in case of failures, the system will backtrack by restoring ℓ'. In a state, names a and id are unique. As usual, parallel composition in states is associative and commutative.

The semantics of a smart contract program is defined by means of transition relation $\mathsf{S} \xrightarrow[\psi]{\mu} \mathsf{S}'$, where μ may be either empty or $\checkmark$ or fail, and ψ is a formula. The class declarations are kept implicit in the transition relation. The reader may find the formal definition of $\xrightarrow[\psi]{\mu}$ in Figure 2.

States may evolve in two ways: either by a transition of the blockchain-statement or by a transition of a user.

The blockchain-statement may evolve because of an empty-labelled statement transition – rule [SC-MOVE] – or because of a method invocations of a smart contract – see rules [INVK] and [INVK-SELF]. In this last case, the state must contain the smart contract whose method is invoked in the label of the transition. Users have a behaviour, which is modelled by a runtime statement, and evolve *concurrently* either with empty-labelled transitions – rule [ID-MOVE] – or with self-invocations – rule [INVK-ID] (therefore a user cannot invoke another user, *e.g.* user interactions are always mediated by a smart contract). When the blockchain-statement is `0`, a user may invoke a smart contract's method – rule [INVK-SC]. This is the only way to start a *blockchain transaction* and, in this case, in order to return the result to the caller, method's body is suffixed with user's name. The transaction terminates either successfully returning a not-fail value to the user that triggered it – rule [END-OK] – or with a failure – rule [END-FAIL]. In the first case, the smart contracts that were involved in the transaction are committed – rule [CMT] –, e.g. their current local memory is saved; in the second case the smart contracts backtrack – rule [BKT] – e.g. their current memory is deleted and the current one becomes the last memory that has been committed. Backtrack and commit are defined by the auxiliary functions $backtk(\cdot)$ and $commit(\cdot)$ in Figure 2.

We conclude by noticing that blockchain-statements are executed *sequentially* and in a *deterministic way* (because they originated in smart contracts' methods). On the contrary, users' method are performed *concurrently* and are *nondeterministic* (because the operator + may occur in their code).

Example 1. We use a simple example to illustrate the technicalities we have introduced. The example is about garbage collection and defines the interactions between a citizen and a smart bin. In particular, a citizen gets rid of the garbage in two ways: either throwing the garbage bags into a smart bin or dumping it (littering the street, for instance). The throwing in the bin is performed by invoking a method "`throw`"; this invocation returns a natural value that corresponds to a cash prize for having behaved well. If the garbage bag is dumped, the prize is 0. Every two garbage bags, the cash owned by the citizen is deposited in his bank account (in order to reduce his overall garbage taxes). The classes of the citizen and of the smart bin are displayed in Figure 3 (notice that the citizen is a user, while the bin is a smart contract). For simplicity, the management of failures has been removed by the codes of Figure 3. The smart bin class has a method `throw` whose behaviour depends on the value of the field `h`. When `h` is 0, two bags can be taken: the field `h` is set to 1 and a prize `k` is returned to the citizen, subtracting it from the field `a` that represents the money hold by the bin. When the second bag arrives, a value $\mathtt{k}'$ is returned to the citizen (notice that `k` may be different from $\mathtt{k}'$), after having re-charged the field `a` and asked to the `truck` to empty the bin, re-setting `h` to 0.

Let us discuss a possible state and its transitions. Let $man \in \mathtt{Citizen}$, $bin \in \mathtt{Garbage\_bin}$, $bank \in \mathtt{Bank}$ and $truck \in \mathtt{Truck}$. We assume that `Bank`

```
Citizen = (
   Nat v1, v2, tmp ;
   Nat behaviour(Id bin, Id bank) =
         this.v1 = bin.throw() ; ( this.v2 = bin.throw() ; CONT
                                 + this.v2 = 0 ; CONT )
         + this.v1 = 0 ; ( this.v2 = bin.throw() ; CONT
                         + this.v2 = 0 ; CONT )
        where CONT = this.tmp = bank.deposit(v1+v2); this.v1 = 0; this.v2 = 0;
                     this.tmp = this.behaviour(bin,bank); return this.tmp
)

Garbage_bin = (
   Nat h, a, tmp ;
   Nat throw() =
        if (this.h == 0) then this.h = 1 ; this.a = this.a - k ; return k
        else this.a = this.a + bank.withdraw(k + k'); this.h = 0 ;
             this.tmp = truck.empty(); this.a = this.a - k'; return k'
)
```

Fig. 3. The citizen and garbage bin classes

has methods `deposit` and `withdraw` (with the obvious meanings); `Truck` has a method `empty` that empties the bin and returns 0. Let also

$$\begin{aligned}\ell_{\mathtt{m}} &= [\mathtt{v}_1 \mapsto 0, \mathtt{v}_2 \mapsto 0, \mathtt{tmp} \mapsto 0]\\ \ell_{\mathtt{b}} &= [\mathtt{h} \mapsto 0, \mathtt{a} \mapsto \mathtt{k}' + \mathtt{k}, \mathtt{tmp} \mapsto 0]\\ s &= \mathit{man}.\mathtt{tmp} = \mathit{man}.\mathtt{behaviour}(\mathit{bin}, \mathit{bank})\,;\mathsf{return}\ 0\\ \mathsf{S} &= \mathit{man}(\ell_{\mathtt{m}}, s) \mid \mathit{bin}(\ell_{\mathtt{b}} \cdot \ell_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid \mathtt{0}\end{aligned}$$

We have

$$\mathsf{S} \underset{\mathtt{true}}{\longrightarrow} \mathit{man}(\ell_{\mathtt{m}}, s') \mid \mathit{bin}(\ell_{\mathtt{b}} \cdot \ell_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid \mathtt{0}$$

where $s' = s_{\mathtt{bh}}\,;\mathit{man}.\mathtt{tmp} = \bullet\,;\mathsf{return}\ 0$ and $s_{\mathtt{bh}}$ is the body of `behaviour` with the instantiation $\{^{\mathit{bin},\mathit{bank},\mathit{man}}/_{\mathtt{bin},\mathtt{bank},\mathtt{this}}\}$. In this state, $s_{\mathtt{bh}}$ may evolve either by invoking $\mathit{bin}.\mathtt{throw}()$ (throwing the garbage into the bin) or by updating $\mathtt{v}_1$ to 0 (illegal dumping of garbage). Let us discuss the first alternative, which is more interesting. Therefore, let $s'' = s'_{\mathtt{bh}}\,;\mathit{man}.\mathtt{tmp} = \bullet\,;\mathsf{return}\ 0$, where

$$s'_{\mathtt{bh}} = \mathit{man}.\mathtt{v1} = \bullet\,;(\mathit{man}.\mathtt{v2} = \mathit{bin}.\mathtt{throw}()\,;\mathtt{CONT} + \mathit{man}.\mathtt{v2} = \mathtt{0}\,;\mathtt{CONT})$$

Then we have the following transitions (x_1 and x_2 are two fresh variables for tracing the choices that has been done):

$$\begin{aligned}&\mathit{man}(\ell_{\mathtt{m}}, s') \mid \mathit{bin}(\ell_{\mathtt{b}} \cdot \ell_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid \mathtt{0}\\ &\quad\underset{\mathtt{true}}{\longrightarrow}\underset{x_1=1}{\longrightarrow} \mathit{man}(\ell_{\mathtt{m}}, s'_{\mathtt{bh}}) \mid \mathit{bin}(\ell_{\mathtt{b}} \cdot \ell_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid s_{\mathtt{thw}}\,;\mathit{man}\\ &\quad\underset{\mathtt{true}}{\longrightarrow}^{4} \mathit{man}(\ell_{\mathtt{m}}, s'_{\mathtt{bh}}) \mid \mathit{bin}(\ell'_{\mathtt{b}} \cdot \ell_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid \mathit{man}[k]\\ &\quad\underset{\mathtt{true}}{\overset{\checkmark}{\longrightarrow}} \mathit{man}(\ell'_{\mathtt{m}}, s'_{\mathtt{bh}}\{^{k}/_{\bullet}\}) \mid \mathit{bin}(\ell'_{\mathtt{b}} \cdot \ell'_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid \mathtt{0}\\ &\quad\underset{\mathtt{true}}{\longrightarrow}\underset{x_2=1}{\longrightarrow} \mathit{man}(\ell'_{\mathtt{m}}, s''_{\mathtt{bh}}) \mid \mathit{bin}(\ell'_{\mathtt{b}} \cdot \ell'_{\mathtt{b}}) \mid \mathit{bank}(\ell \cdot \ell) \mid \mathit{truck}(\ell' \cdot \ell') \mid s_{\mathtt{thw}}\,;\mathit{man}\end{aligned}$$

where $s_{\mathtt{thw}}$ is the instance of the body of `throw` with the name *bin* for this; $\ell'_{\mathtt{m}} = \ell_{\mathtt{m}}[\mathtt{v}_1 \mapsto \mathtt{k}]$, $\ell'_{\mathtt{b}} = \ell_{\mathtt{b}}[\mathtt{h} \mapsto 1, \mathtt{a} \mapsto \mathtt{k}']$ and $s''_{\mathtt{bh}} = \mathit{man}.\mathtt{v2} = \bullet\,;\mathtt{CONT}$. The continuation is omitted.

3 The open semantics and the analysis model

A decentralized application is never a closed system: smart contract's methods can always be invoked not only by users and the smart contracts designed to interact with them, but also by unknown actors. Therefore, to study properties of our systems, we need to analyse open configurations. In particular, we need to reason on invocations without any knowledge of the actual parameters of the caller. A standard solution of this problem is to use *symbolic variables* – see Figure 4 –, i.e. extending values with (unbound) variables and admitting that operators return terms, such as $x + 1$, in addition to integers and actor names. Therefore the evaluation function $[\![e]\!]_{\alpha,\ell}$ may now return terms with symbolic variables and it follows that actor's fields may also record terms with symbolic variables.

The extensions of runtime statements transitions in Figure 1 and of state transitions in Figure 2 are given in Figure 4. As regards runtime statements, the rules [IF-TRUE] and [IF-FALSE] are replaced by [IF-OPEN-TRUE] and [IF-OPEN-FALSE] where the formulas of transitions report whether the guard is true ($\neq 0$) or false ($= 0$). These formulas and those of the rule [CHOICE] will enable the analysis of smart contract systems in Section 4.

As regards the open state transition, we extend the rules of Figure 2 with those for invoking a method of a unspecified smart contract – rules [INVK-OPEN] and [INVK-OPEN-ID]. We discuss the former, the latter one is similar. The function $\bullet \notin s''$ returns true if $\bullet$ does not occur in the runtime statement s'' (which is always the case when a method body is istantiated). This expedient is used to select in $s''\mathtt{;}a.\mathtt{f} = \bullet\mathtt{;}s', a, \ell$ the prefix representing the instance of method's body and to drop it because we don't want to analyze behaviours of unspecified actors. Henceforth, the rule delivers to the continuation either a symbolic variable or fail, therefore covering every possible output of the invocation.

Rule [INPUT-OPEN-SC] defines the invocation of a smart contract method by a hypothetical user. In this case, actual parameters are all fresh symbolic variables and the instance of method body is suffixed by the name of the hypothetical user. According to the returned value is fail or not, we will have a backtrack – rule [END-FAIL-OPEN] – or a commit – rule [END-OPEN] –, respectively.

The open transition system in Figure 4 defines a *model* that is a tuple $(\mathcal{S}, \mathsf{S}_0, \mathcal{T})$, where $\mathcal{S}$ is a non-empty set of *states*, $\mathsf{S}_0 \in \mathcal{S}$ is the initial state, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S} \times \Theta \times \Psi$ is the set of labelled transitions. The set Θ is the collection of labels $\{\varepsilon, \mathsf{fail}, \checkmark, v = a'.\mathtt{m}(\overline{v}), id : \overline{z}\}$ (ε represents the empty label) while Ψ is a set of formulae. As usual $\langle \mathsf{S}_1, \mathsf{S}_2, \mu, \psi \rangle \in \mathcal{T}$ is abbreviated into $\mathsf{S}_1 \xrightarrow[\psi]{\mu} \mathsf{S}_2$.

Actually, in Section 4 we use a slightly different model than the foregoing one, that we call *analysis model*. In the forthcoming analysis we need to deal with the constant fail. To this aim, in order to remain in Presburger arithmetics, we decided to encode fail by extending the *Nat* type to *Integers*. Henceforth fail is encoded by -1 and we use a function $|\psi|$ replacing every occurrence of fail with -1 and turning every actor name into a (global) integer variable. We also extend labels Θ with a new label, written (x), which is meant to expose in the

Symbolic values:

$$v ::= \ \ldots \ \mid \ x \ \mid \ v \ \mathsf{op} \ v \qquad \text{(symbolic values)}$$

Open runtime statements transition $\overset{\mu}{\underset{\psi}{\longmapsto}}$:

$$\text{[IF-OPEN-TRUE]}\ \frac{[\![e]\!]_{\alpha,\ell} = v}{\alpha : \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s'; s'', \ell \underset{v\neq 0}{\longmapsto} s; s'', \ell} \qquad \text{[IF-OPEN-FALSE]}\ \frac{[\![e]\!]_{\alpha,\ell} = v}{\alpha : \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s'; s'', \ell \underset{v=0}{\longmapsto} s'; s'', \ell}$$

Open state transition $\overset{\mu}{\underset{\psi}{\longrightarrow}}$ (μ may be empty, $\checkmark$, fail, $a.\mathtt{m}(\overline{v})$, or $id : \overline{z}$; ψ is a formula):

$$\text{[INVK-OPEN]}\ \frac{a : s, \ell \overset{a'.\mathtt{m}(\overline{v'})}{\underset{\mathtt{true}}{\longmapsto}} s''; a.\mathtt{f} = \bullet; s', \ell \quad \bullet \notin s'' \quad v \text{ either } x \text{ fresh or } \mathsf{fail}}{\mathsf{S} \mid a(\ell \cdot \ell') \mid s \overset{v=a'.\mathtt{m}(\overline{v'})}{\underset{a' \notin \mathsf{S}}{\longrightarrow}} \mathsf{S} \mid a(\ell \cdot \ell') \mid (a.\mathtt{f} = \bullet; s')[v]}$$

$$\text{[INVK-OPEN-ID]}\ \frac{id : s, \ell \overset{a.\mathtt{m}(\overline{v'})}{\underset{\mathtt{true}}{\longmapsto}} s''; id.\mathtt{f} = \bullet; s', \ell \quad \bullet \notin s'' \quad v \text{ either } x \text{ fresh or } \mathsf{fail}}{\mathsf{S} \mid id(\ell, s) \overset{v=a.\mathtt{m}(\overline{v'})}{\underset{a \notin \mathsf{S}}{\longrightarrow}} \mathsf{S} \mid id(\ell, (id.\mathtt{f} = \bullet; s')[v])}$$

$$\text{[INPUT-OPEN-SC]}\ \frac{id, \overline{z} \text{ fresh} \quad a \in \mathtt{C} \quad \mathtt{m}(\overline{x}) = s_{\mathtt{m}} \in \mathtt{C} \quad s'' = s_{\mathtt{m}}\{^{\overline{z},a}/_{\overline{x},\mathsf{this}}\}}{\mathsf{S} \mid a(\ell \cdot \ell) \mid 0 \overset{id:\overline{z}}{\underset{id \notin \mathsf{S}}{\longrightarrow}} \mathsf{S} \mid a(\ell \cdot \ell) \mid s''; id}$$

$$\text{[END-OPEN]}\ \frac{v \neq \mathsf{fail}}{\mathsf{S} \mid id[v] \overset{\checkmark}{\underset{id \notin \mathsf{S}}{\longrightarrow}} commit(\mathsf{S}) \mid 0} \qquad \text{[END-FAIL-OPEN]}\ \mathsf{S} \mid id[\mathsf{fail}] \overset{\mathsf{fail}}{\underset{id \notin \mathsf{S}}{\longrightarrow}} backtk(\mathsf{S}) \mid 0$$

Fig. 4. The open semantics of `scl`.

label the fixing of the variable x by rule [FIX] of Figure 1. In particular, in the analysis model

- if $\mathsf{S} \overset{\mu}{\underset{\psi}{\longrightarrow}} \mathsf{S}'$ follows from the open semantics *without using the rule* [FIX] then the analysis model has $\mathsf{S} \overset{\mu}{\underset{|\psi|}{\longrightarrow}} \mathsf{S}'$;
- if $\mathsf{S} \underset{\psi}{\longrightarrow} \mathsf{S}'$ follows from the open semantics *using the rule* [FIX] and x is the fresh variable that has been introduced then the analysis model has $\mathsf{S} \overset{(x)}{\underset{|\psi|}{\longrightarrow}} \mathsf{S}'$.

An (analysis) model is *finite* if the set of states $\mathcal{S}$ is finite; it is a *Presburger* model if the set of conditions Ψ and the actual parameters used in Θ range respectively over Presburger formulas and Presburger expressions. We recall that Presburger arithmetics is the decidable subset of classical first order logic over integer numbers where the only predicates allowed are equality and (strict) inequality and the only function symbols are addition between integer expressions and multiplication of an integer expression by a constant.

Example 2. In Figure 5 we illustrate the analysis model of a citizen and a garbage bin discussed in Example 1. In order to have a more compact picture, we have

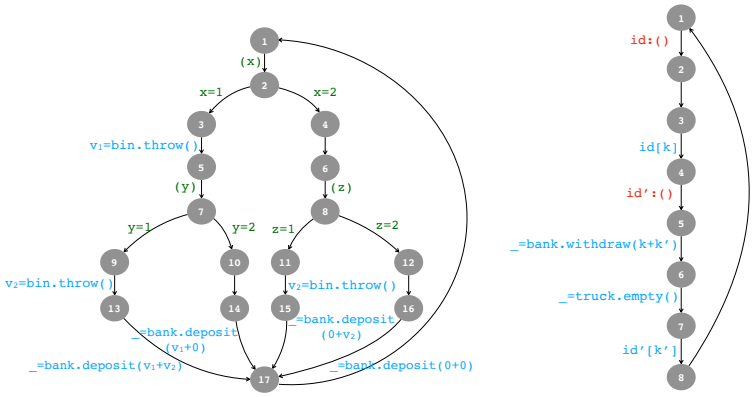

Fig. 5. The models of the citizen and the bin

collapsed empty-labelled transitions. We have also used a simple hack to have finite models: we have added an empty labelled transition from state 17 to state 1, even if they are not exactly equal as systems states. This is because the stack of the citizen grows at every call since we do not optimize tail recursion in the transition systems of Figure 2. However, our analyzer always performs these optimizations. We also remark that the two models are also Presburger models.

Due to lack of space, Figure 5 does not report the interactions between the citizen, the garbage bin and other actors. The complete model has a large number of states because of all possible interactions. In particular, every time the citizen wants to throw out her garbage, she can either succeed immediately — and in that case an empty labelled move is performed by the system via the [INVK-SC] rule — or she can be pre-empted by an unknown *id* via the [INPUT-OPEN-SC] rule.

4 Observables and strategies

A user can behave in different ways and obtain different results because of the choice operator + that may occur in its code. In this section we study users' behaviours and how choices influence results in a sensible way.

Definition 1. *Let* $(\mathcal{S}, \mathsf{S}_0, \mathcal{T})$ *be a* `scl` *analysis model. An* objective function ω *maps labels* μ *of transitions* $\mathsf{S} \xrightarrow[\psi]{\mu} \mathsf{S}'$ *in* $\mathcal{T}$ *to some integer expression. Given a computation* $\mathsf{S}_1 \xrightarrow[\psi_1]{\mu_1} \cdots \xrightarrow[\psi_n]{\mu_n} \mathsf{S}_{n+1}$*, its* ω-observation *is* $\sum_{1 \le i \le n} \omega(\mu_i)$.

In Example 1, a citizen is interested into maximizing the amount of cash-back that is saved in the bank, i.e. its objective function called ω_C, is defined as follows

$$\omega_C(\mu) = \begin{cases} v_1 + v_2 & \text{if } \mu = \_ = \texttt{bank.deposit}(v_1 + v_2) \\ 0 & \text{otherwise} \end{cases}$$

We notice that the result of an objective function may be *a term containing symbolic names*. (As a consequence our analysis will enable programmers to design smart contracts with instances of symbolic names that support "optimal strategies" – see below.) Contrary to citizens, the city hall is interested on minimizing the amount of garbage dumped or, equivalently, to maximize the amount of garbage thrown in the bins. Thus its objective function ω_H is defined as follows

$$\omega_H(\mu) = \begin{cases} 1 & \text{if } \mu = \_ = \texttt{bin.throw()} \\ 0 & \text{otherwise} \end{cases}$$

A *strategy* is a function that determines the transition to perform in states where the next statement to execute is a choice. Since these decisions are taken by users, which only have a *partial understanding* of the state (every user has full visibility of its own state and of smart contracts' states, they cannot access the internal state of other users), in general, a user can only devise "sub-optimal" strategies. A strategy is *optimal* with respect to an objective function ω, if, for any other strategy, all the possible future computations (which follow by the behaviour of the other users involved) cannot yield a higher ω-observation. Users that, at any choice point, try to maximize some *objective function* are called *rational*. In the rest of the section we are interested into rational users.

In general we cannot expect the existence of optimal strategies, nor we can expect that a strategy that is optimal for a given objective function is also optimal for another objective function. For example, if we just analyze the model of our citizen in isolation (see Figure 5), we can only deduce the following facts about the pair of objective functions $\langle \omega_C, \omega_H \rangle$, i.e. the first element observes the cash-back (according to the citizen objective function) and the second one the number of calls to `throw()` (according to the city hall objective function):

1. the strategy that dumps the garbage twice at each execution 1–17 yields the pair $\langle 0, 0 \rangle$;
2. the two strategies that dump the garbage once yield the pairs $\langle v_1, 1 \rangle$ and $\langle v_2, 1 \rangle$ respectively, for some v_1, v_2;
3. the strategy that never dumps the garbage yield the pair $\langle v_1 + v_2, 2 \rangle$, for some v_1, v_2.

Note that v_1 and v_2 are not fixed: a garbage bin may return different values at every invocation of `throw()`, depending on its own internal logic and on the interaction with other actors. From the point of view of the city hall, the optimal strategy for the citizen is clearly the last one. From the point of view of the citizen, instead, there is no optimal strategy: it depends on the behaviour of the garbage bin that is unknown since we are analyzing the citizen in isolation.

Here is where smart contracts enter into the picture. A smart contract is used to regulate interaction between (human) users. In many real world examples, the smart contract is designed so that the system obtained composing humans with contracts has two relevant properties: (*i*) every user has an optimal strategy and (*ii*) the optimal users' strategies also maximizes the objective function of the smart contract programmer. In Example 1, this second function is exactly ω_H,

that is the programmer aims at minimizing the garbage dumped by citizens. If we combine the citizen and the smart contract and we analyze again the model obtained, we realize that the garbage bin sometimes pays back k coins and sometimes k' coins. In particular, the strategy of point 1 always yield $\langle 0, 0\rangle$, both strategies of point 2 sometimes yield $\langle k, 1\rangle$ and sometimes $\langle k', 1\rangle$, while the strategy of point 3 can yield any pair of observations among $\langle k+k, 2\rangle$, $\langle k+k', 2\rangle$, $\langle k'+k, 2\rangle$, $\langle k'+k', 2\rangle$. It follows that the strategy of point 1 is clearly not optimal for the citizen, but the remaining three are incomparable because, for example, getting a k' from one strategy frequently can be better than getting a $k+k$ from another strategy.

Our analysis, however, suggests a simple solution to the programmer: if the programmer chooses $k = k' > 0$, then the four strategies now yield the pairs $\langle 0, 0\rangle, \langle k, 1\rangle, \langle k, 1\rangle, \langle 2k, 2\rangle$. In this case, the last strategy is optimal both for the citizen and for the city hall. The same result can be obtained picking any value for k and k' such that $0 < k < 2k' < 4k$, so that getting two cashbacks is always better than getting just one of them.

In the rest of the section, the analysis models will be *acyclic* and *finite*, namely models such that the transition relation has always finite maximal computations. These models can be obtained by unfolding cyclic models up to a certain depth. This makes sense in the context of smart contracts where the executions are always bound in the number of steps by some quantity called *gas* that is bought by the caller (a user) with the virtual currency of the blockchain [6]. It also makes sense in examples like Example 1 where the model is one big cycle; therefore it is sufficient to maximize the objective function over just one iteration of the loop.

4.1 Automatic analysis

We use an automatic technique to verify whether an analysis model retains optimal strategies for some objective function or whether maximizing some objective function – e.g. the cash-back to the citizen – implies the maximization of other objective functions – e.g. the amount of garbage thrown into smart bins. Given an objective function, our technique derives a first order logic formula specifying every possible observation with respect to unknown symbolic inputs and to the internal choices of every agent.

Let $\mathcal{Q}$ be the following map from labels to a possibly empty sequence of quantifiers:

$$\begin{array}{rrr} \mathcal{Q}(\texttt{(}x\texttt{)}) = \exists x & \qquad & \mathcal{Q}(x = a.\mathtt{m}(\overline{v})) = \forall x \\ \mathcal{Q}(id : \overline{z}) = \forall id \forall \overline{z} & & \mathcal{Q}(\mu) = \varepsilon \qquad \text{(otherwise)} \end{array}$$

Let also $\mathcal{M} = (\mathcal{S}, \mathsf{S}_0, \mathcal{T})$ be finite and acyclic and ω be an objective function. The *characteristic formula* $(\!|\ \mathsf{S}_0\ |\!)^{\mathcal{M}}_{\omega}$ is the first-order logic formula defined as follows

– when there is at least one transition $\mathsf{S} \xrightarrow[\psi]{\mu} \mathcal{S}' \in \mathcal{T}$:

$$(\!|\, \mathsf{S} \,|\!)^{\mathcal{M}}_{\omega} = \bigvee_{\mathsf{S} \xrightarrow[\psi]{\mu} \mathcal{S}' \in \mathcal{T}} \mathcal{Q}(\mu)\Big(\psi \wedge \mathcal{O}(\mathsf{S}, \{\omega(\mu)\}) \wedge (\!|\, \mathsf{S}' \,|\!)^{\mathcal{M}}_{\omega}\Big) \quad ;$$

– when there is no μ, ψ, S' such that $\mathsf{S} \xrightarrow[\psi]{\mu} \mathcal{S}' \in \mathcal{T}$ (S is final):

$$(\!|\, \mathsf{S} \,|\!)^{\mathcal{M}}_{\omega} = \top$$

where $\top$ is the proposition "true".

The binary predicate $\mathcal{O}(\mathsf{S}, \mathcal{A})$ in the characteristic formula is to be read as follows: there exists a computation originating from S that eventually leaves the state S observing one element of the set $\mathcal{A}$. For example, $\mathcal{O}(13, \{v_1 + v_2\})$ means that there will be a transition from state 13 where the user observes $v_1 + v_2$; $\mathcal{O}(15, \{2\}) \vee (\mathcal{O}(12, \{v_1, v_2\})$ means that either state 15 will be reached and left observing 2, or the state 12 will be reached and left observing either v_1 or v_2.

In order to ease the reading, we simplify the characteristic formula as follows:

– $G \wedge \mathcal{O}(\mathsf{S}, \{0\})$ is simplified to G because observing 0 is useless for maximizing an objective function;
– $G \wedge \top$ is simplified to G;
– $\forall x.G$ is simplified to G when x does not occur in G (similarly for $\exists$).

After these simplifications, the characteristic formula for the objective function ω_C of the citizen (we remove the empty-labelled transition from state 17 to state 1 in Figure 5) is:

$$\begin{aligned}(\!|\, 1 \,|\!)^{\mathcal{M}}_{\omega_C} = \; \exists x \Big[\;\; & x = 1 \;\wedge\; \forall v_1 \exists y \; \Big((y = 1 \wedge \forall v_2 \, \mathcal{O}(13, \{v_1 + v_2\})) \vee (y = 2 \wedge \mathcal{O}(14, \{v_1 + 0\})) \Big) \\ & \vee \;\; x = 2 \;\wedge \exists z \Big(z = 1 \wedge \forall v_2 \, \mathcal{O}(15, \{0 \;\; + v_2\})) \vee \big(z = 2 \wedge \mathcal{O}(16, \{0 \;\; + 0\})\big) \Big) \;\; \Big] \end{aligned} \quad (1)$$

The characteristic formula for the objective function ω_H of the city hall instead is:

$$\begin{aligned}(\!|\, 1 \,|\!)^{\mathcal{M}}_{\omega_H} = \exists x \Big[\;\; & x = 1 \;\wedge\; \Big(\mathcal{O}(3, \{1\}) \wedge \; \exists y ((y = 1 \wedge \mathcal{O}(9, \{1\})) \vee y = 2) \Big) \\ & \vee \;\; x = 2 \;\wedge\; \exists z \Big((z = 1 \wedge \mathcal{O}(11, \{1\})) \vee z = 2 \Big) \;\; \Big] \end{aligned} \quad (2)$$

4.2 Quantifier elimination

When the analysis model is Presburger, the characteristic formula belongs to the extension of Presburger arithmetics with the observation predicate $\mathcal{O}(\mathsf{S}, \mathcal{A})$. It turns out that this fragment of first order logic can be decided via quantifier elimination [13]: at every step the formula is rewritten so that each innermost quantifier is existential, and then that quantified formula is replaced with a logically equivalent one where the existentially bound variable no longer occurs.

As a special bonus, the sets $\mathcal{A}$ of observation predicates can be rewritten in the elimination step in such a way that at the end we can recover from the quantifier free formula a set of polytopes that describe the value of all possible variables in every observable state that can be reached, together with the observation performed in that state. It is then easy to compare different strategies observing the value taken by the objective function when its input ranges over the polytope.

The quantifier elimination algorithm starts rewriting an innermost quantified subformula into the normal form

$$\exists x.\bar{l} \leq kx \wedge kx \leq \overline{u} \wedge \bigwedge_i \natural_i \mathcal{O}(\mathsf{S}_i, \mathcal{A}_i)$$

where $x \notin \bar{l}, \overline{u}$ and $\natural F$ stands for either F or $\neg F$. Then it replaces the formula with its logically equivalent one

$$(\bigwedge_{l_i \in \bar{l}} \bigwedge_{u_j \in \overline{u}} l_i \leq u_j) \wedge \bigwedge_i \natural_i \mathcal{O}(\mathsf{S}_i, \{a \in \mathcal{A}_i \mid \bar{l} \leq kx \wedge kx \leq \overline{u}\})$$

Afterwards the algorithm loops on another innermost quantifier until no quantifiers are left. Special care is required to avoid simplifying $\top \vee F$ into $\top$ and $\bot \wedge F$ into $\bot$ to avoid loosing the polytopes recoverable from F.

Example 3. We illustrate the quantifier elimination technique on the characteristic formula (1). In the following, the underlined formulas are the next ones to be simplified. For the sake of readability, we shorten expressions as $0 + 0$ into 0 and $v + 0$ into v.

$$\begin{aligned}
(\!|\, 1 \,|\!)^{\mathcal{M}}_{\omega_C} &= \exists x\Big[x = 1 \ \wedge\ \forall v_1 \exists y\ \big((y = 1 \wedge \underline{\forall v_2\, \mathcal{O}(13, \{v_1 + v_2\})}) \vee (y = 2 \wedge \mathcal{O}(14, \{v_1\}))\big) \\
&\qquad \vee\ x = 2 \ \wedge \exists z\big(z = 1 \wedge \underline{\forall v_2\, \mathcal{O}(15, \{v_2\})}\big) \vee \big(z = 2 \wedge \mathcal{O}(16, \{0\})\big)\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \forall v_1 \exists y\ \big((y = 1 \wedge \underline{\neg\exists v_2\, \neg\mathcal{O}(13, \{v_1 + v_2\})}) \vee (y = 2 \wedge \mathcal{O}(14, \{v_1\}))\big) \\
&\qquad \vee\ x = 2 \ \wedge \exists z\big(z = 1 \wedge \underline{\neg\exists v_2\, \neg\mathcal{O}(15, \{v_2\})}\big) \vee \big(z = 2 \wedge \mathcal{O}(16, \{0\})\big)\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \forall v_1 \underline{\exists y\ \big((y = 1 \wedge \neg\neg\mathcal{O}(13, \{v_1 + v_2\})) \vee (y = 2 \wedge \mathcal{O}(14, \{v_1\}))\big)} \\
&\qquad \vee\ x = 2 \ \wedge \underline{\exists z\big(z = 1 \wedge \neg\neg\mathcal{O}(15, \{v_2\})\big) \vee \big(z = 2 \wedge \mathcal{O}(16, \{0\})\big)}\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \forall v_1 \big(\underline{\exists y(y = 1 \wedge \mathcal{O}(13, \{v_1 + v_2\}))} \vee \underline{\exists y(y = 2 \wedge \mathcal{O}(14, \{v_1\}))}\big) \\
&\qquad \vee\ x = 2 \ \wedge \big(\underline{\exists z\big(z = 1 \wedge \mathcal{O}(15, \{v_2\})\big)} \vee \underline{\exists z\big(z = 2 \wedge \mathcal{O}(16, \{0\})\big)}\big)\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \underline{\forall v_1 \big(\mathcal{O}(13, \{v_1 + v_2 \mid y = 1\}) \vee \mathcal{O}(14, \{v_1 \mid y = 2\})\big)} \\
&\qquad \vee\ x = 2 \ \wedge\ \big(\mathcal{O}(15, \{v_2 \mid z = 1\}) \vee \mathcal{O}(16, \{0 \mid z = 2\})\big)\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \underline{\neg\exists v_1 \big(\neg\mathcal{O}(13, \{v_1 + v_2 \mid y = 1\}) \wedge \neg\mathcal{O}(14, \{v_1 \mid y = 2\})\big)} \\
&\qquad \vee\ x = 2 \ \wedge\ \big(\mathcal{O}(15, \{v_2 \mid z = 1\}) \vee \mathcal{O}(16, \{0 \mid z = 2\})\big)\Big] \\
&\iff \exists x\Big[x = 1 \ \wedge\ \big(\mathcal{O}(13, \{v_1 + v_2 \mid y = 1\}) \vee \mathcal{O}(14, \{v_1 \mid y = 2\})\big) \\
&\qquad \vee\ \underline{x = 2 \ \wedge\ \big(\mathcal{O}(15, \{v_2 \mid z = 1\}) \vee \mathcal{O}(16, \{0 \mid z = 2\})\big)}\Big] \\
&\iff \underline{\exists x\Big[x = 1 \wedge \mathcal{O}(13, \{v_1 + v_2 \mid y = 1\}) \vee x = 1 \wedge \mathcal{O}(14, \{v_1 \mid y = 2\})} \\
&\qquad \underline{\vee\ x = 2 \wedge \mathcal{O}(15, \{v_2 \mid z = 1\}) \vee x = 2 \wedge \mathcal{O}(16, \{0 \mid z = 2\})\Big]} \\
&\iff \underline{\exists x\Big[x = 1 \wedge \mathcal{O}(13, \{v_1 + v_2 \mid y = 1\})\Big] \vee \exists x\Big[x = 1 \wedge \mathcal{O}(14, \{v_1 \mid y = 2\})\Big]} \\
&\qquad \underline{\vee\ \exists x\Big[x = 2 \wedge \mathcal{O}(15, \{v_2 \mid z = 1\})\Big] \vee \exists x\Big[x = 2 \wedge \mathcal{O}(16, \{0 \mid z = 2\})\Big]} \\
&\iff \mathcal{O}(13, \{v_1 + v_2 \mid x = 1 \wedge y = 1\}) \vee \mathcal{O}(14, \{v_1 \mid x = 1 \wedge y = 2\}) \\
&\qquad \vee\ \mathcal{O}(15, \{v_2 \mid x = 2 \wedge z = 1\}) \vee \mathcal{O}(16, \{0 \mid x = 2 \wedge z = 2\})
\end{aligned}$$

Applying the above technique to the equation (2), we derive

$$\Big(\mathcal{O}(3, \{1 \mid x = 1\}) \wedge \mathcal{O}(9, \{1 \mid x = 1 \wedge y = 1\})\Big) \vee \mathcal{O}(11, \{1 \mid x = 2 \wedge z = 1\})$$

We notice that, in the case of this last formula, the best strategy to maximize ω_H is to choose $x = 1 \wedge y = 1$ that yields the observation $1 + 1 = 2$. More precisely, the strategy consists in picking the branch 1 of the rule [CHOICE] in the state $\ldots +^x \ldots$ and the branch 1 in the state $\ldots +^y \ldots$. With an abuse of notation, we will indicate strategies as conjunctions $\bigwedge_{i \in 1..n} x_i = k_i$.

Let us discuss strategies for ω_C. According to the formula

$$\begin{array}{c} \mathcal{O}(13, \{v_1 + v_2 \mid x = 1 \wedge y = 1\}) \vee \mathcal{O}(14, \{v_1 + 0 \mid x = 1 \wedge y = 2\}) \\ \vee\ \mathcal{O}(15, \{0\ + v_2 \mid x = 2 \wedge z = 1\}) \vee \mathcal{O}(16, \{0\ + 0 \mid x = 2 \wedge z = 2\}) \end{array}$$

there is no best strategy to maximize ω_C because, for example, by choosing $x = 1 \wedge y = 1$, one can observe any value in the set $\{v_1 + v_2\}$. In fact, putting the citizen in parallel with the garbage bin, the formal parameters v_1 and v_2 can only be instantiated with either k or k', according to the interleaving of the moves of the citizen with the other actors. In particular, running the analysis again on the larger model – due to the interleaving –, we obtain the characteristic formula:

$$\begin{array}{ll} & \mathcal{O}(\{k + k \mid x = 1 \wedge y = 1\}) \ \vee\ \mathcal{O}(\{k + k' \mid x = 1 \wedge y = 1\}) \\ \vee & \mathcal{O}(\{k' + k \mid x = 1 \wedge y = 1\}) \ \vee\ \mathcal{O}(\{k' + k' \mid x = 1 \wedge y = 1\}) \\ \vee & \mathcal{O}(\{k \mid x = 1 \wedge y = 2\}) \ \vee\ \mathcal{O}(\{k' \mid x = 1 \wedge y = 2\}) \\ \vee & \mathcal{O}(\{k \mid x = 2 \wedge z = 1\}) \ \vee\ \mathcal{O}(\{k' \mid x = 2 \wedge z = 1\}) \\ \vee & \mathcal{O}(\{0 \mid x = 2 \wedge z = 2\}) \end{array}$$

In this case, there is still no optimal choice because, for example, by taking $x = 1 \wedge y = 1$ one may observe $k + k$, which may be smaller than k' that is observed for $x = 1 \wedge y = 2$. As already discussed, it is sufficient for the implementor to pick $0 < k < 2k' < 4k$ to force the existence of a best strategy which is $x = 1 \wedge y = 1$ and that coincides with the best strategy from the city hall point of view, as expected.

Remark 1. The standard formal specification languages to verify and specify properties of transition systems are temporal logics [12]. Actually one may use tools that are based on these logics, such as [1], to automatically analyze systems based on smart contracts. In fact, our characteristic formula is actually a compilation of linear-time temporal logics with Presburger constraints, which is decidable [5]. We have preferred the current presentation because it is the one we use in our prototype implementation.

4.3 Implementation

We are terminating the implementation in OCaml of a tool that, given a set of actors, an initial state and an objective function, automatically extracts the

analysis model, computes the characteristic formula and applies quantifier elimination over it.

Initially we hoped to be able to reuse off-the-shelf implementations of Presburger quantifier elimination by dropping the $\mathcal{O}$ predicate and recovering the polytopes from the tools. But the tools we analyzed are unable to spit out the polytopes. In fact, because of the double-exponential complexity of quantifier elimination over the number of alternations of quantifiers, the tools avoid quantifier elimination and rather use model checking or reduction to finite state automata. However, these two techniques can only enumerate the points in the polytope, without providing a closed description of it [9, 8]. Therefore we decided to implement quantifier elimination straight away. We need to evaluate whether our quantifier elimination is is doable in practice on characteristic formulae generated from realistic examples of smart contracts. This is left as future work, once the whole implementation is completed. We will also study the application of other techniques, like temporal logic, to analyse the formal calculus and the models introduced in the paper. To ease these analyses, we will write transpilers from our smart calculus language to existing languages, initially targeting both Solidity and Liquidity.

5 Conclusions

In this paper we have introduced a unified calculus for modelling smart contracts and users, which are the primary actors of decentralized applications. These applications run on blockchain systems and handle and transfer assets of considerable value. We have therefore studied how to regulate by means of smart contracts the interaction between (rational) users that systematically try to maximize their revenue or to minimize losses. This is achieved by expressing the system behaviour as a formula in Presburger arithmetics and solving arithmetic constraints. Our technique is amenable to automated verification and we are currently completing an OCaml implementation.

The analysis of smart contracts for deriving strategies and distilling the most meaningful ones opens unexpected connections between (micro) economy and computer science that deserves further investigations. While this direction of research has been already pointed out in other contributions (see *e.g.* [3]), we believe that there is much work still to be done.

As regards our calculus, several extensions must be considered. First of all, the types must be extended with simple dynamic data types, such as arrays and maps. Then, if we want to model faithfully the (human) users, we need to take into account probabilities because users are not always 100% rational; they may be irrational with some percentage.

Dedication. Cosimo is proud to dedicate this paper to Stefania Gnesi and to the unforgettable Friday afternoons spent together with Alessandro Fantechi. Stefania and Alessandro have been Cosimo's Master Thesis advisors and they first led him to concurrent systems, formal methods and temporal logics. Thank you Stefania!

References

1. Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A model-checking tool for families of services. In *Proc. of Joint 13th IFIP WG 6.1 Int. Conf. on Formal Techniques for Distributed Systems*, volume 6722 of *Lecture Notes in Computer Science*, pages 44–58. Springer, 2011.
2. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proc. of Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.
3. Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, volume 9465 of *Lecture Notes in Computer Science*, pages 142–161. Springer, 2015.
4. Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. Quantitative analysis of smart contracts. In *Proc. of ESOP 2018*, volume 10801 of *Lecture Notes in Computer Science*, pages 739–767. Springer, 2018.
5. Stéphane Demri. Linear-time temporal logics with presburger constraints: an overview. *Journal of Applied Non-Classical Logics*, 16(3-4):311–348, 2006.
6. Ethereum Foundation. Ethereum's white paper. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2014.
7. Ethereum Foundation. Solidity 0.4.24 documentation. `https://solidity.readthedocs.io/en/develop/`, 2019.
8. Vijay Ganesh, Sergey Berezin, and David L. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design*, pages 171–186. Springer, 2002.
9. Christoph Haase. A survival guide to presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, July 2018.
10. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proc. of the Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.
11. Bernhard Mueller. Smashing Ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.
12. Amir Pnueli. The temporal logic of programs. In *Proc. of Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
13. Jeremy Pope. Formalizing constructive quantifier elimination in Agda. In *Proceedings. of MSFP@FSCD 2018.*, volume 275 of *EPTCS*, pages 2–17, 2018.
14. OCamlPro SAS. Welcome to Liquidity's documentation! `http://www.liquidity-lang.org/doc/`, 2019.
15. David Siegel. Understanding the dao attack. *Retrieved June*, 13:2018, 2016.